NAG®

The Numerical Algorithms Group

Mathematics and technology for optimal performance

Briefing on ACML – the AMD Core Math Library

Rob Meyer, President
Numerical Algorithms Group

Results Matter, Trust NAG

Agenda

- Brief introduction to NAG
- AMD / NAG collaboration
- Importance of quality numerical software
- AMD Core Math Library (ACML) contents / status
- Tuning & getting the best results from ACML
- ACML testing for quality
- ACML performance
- Related AMD / NAG collaboration – tuned “libm”
- Lessons & essential tools

Brief Introduction to NAG

- Founded 1970
  - Co-operative software project
- Incorporated 1976
  - Not-for-profit organisation
  - Over 300 member “shareholders”
  - NAG Ltd (UK)
  - NAG Inc (USA)
  - Nihon NAG (Japan)
- ~90 employees
- Main products – Math & Statistical Libraries
  - Also 3D Visualization, Compilers, Tools …

NAG Numerical Libraries

- Over 2500 quality components

- FORTRAN Library
- FORTRAN 90 Library
- C Library
- FORTRAN SMP Library
- Parallel (MPI) Library
- Data Mining Components

NAG HPC Libraries

www.nag.com/numeric.html

NAG Numerical Libraries (continued)

- NAG provides high-level math and stats components
  - Nonlinear equation solvers
  - Summation of series and transformations, FFTs
  - Quadrature
  - ODEs, PDEs and integral equations
  - Approximation and curve and surface fitting
  - Optimization and operations research
  - Dense linear algebra, including LAPACK
  - Sparse linear systems and eigenproblems
  - Special functions

AMD Collaborative Work

- AMD have developed the AMD64 processors (AMD Opteron™, AMD Athlon™ 64)
- AMD64 is NOT compatible with Intel64 (IA64) hence need to develop the full software stack ("ecosystem")
- AMD and NAG collaborating to produce
  - AMD Core Math Library (ACML)
  - libm

AMD Opteron™ and Athlon™ 64 Processors

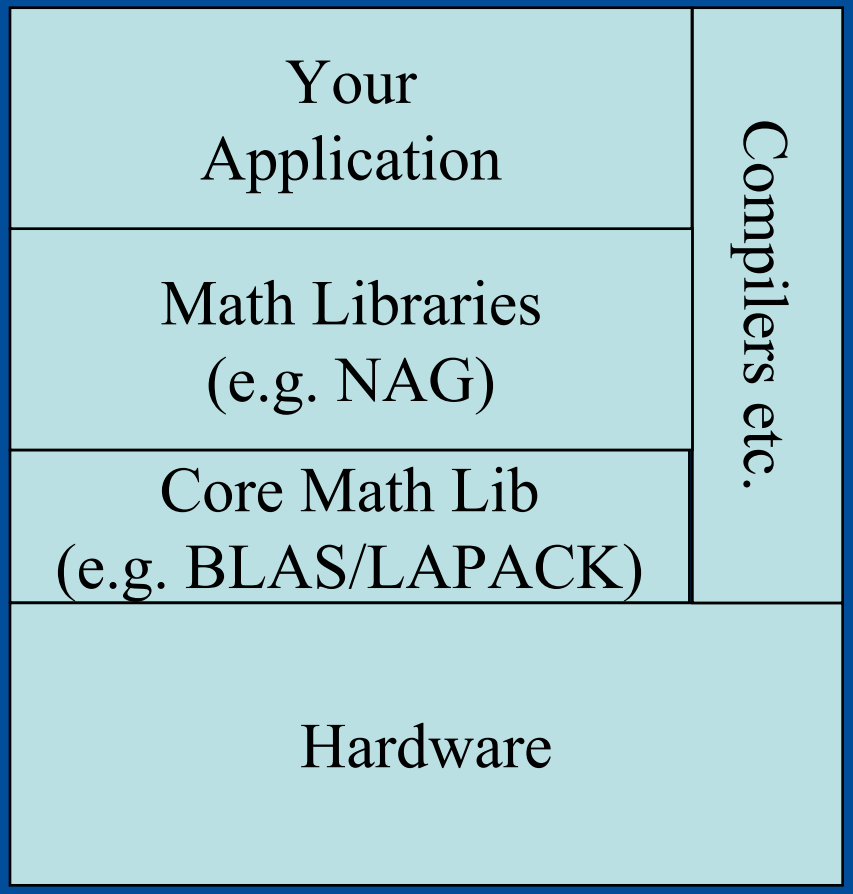
- Run 64-bit and 32-bit code natively (not emulator)
- 32-bit and 64-bit applications can run simultaneously under a 64-bit operating system
- Users can transition from 32-bit to 64-bit computing smoothly – no need to throw away old software
- 32-bit code runs faster on AMD64 than on any previous AMD chip
- AMD Opteron™ was launched April 2003, AMD Athlon™ 64 September 2003

AMD Core Math Library

- NAG collaborating with AMD to produce ACML — the AMD Core Math Library.
  - Announced at LinuxWorld in January 2003
  - Version 1.0 available since June 2003 for download
  - Version 1.5 imminent - November 2003
- Tuned to run well on AMD64 processors
- Extensive test/timing harness

Importance of math libs in development – performance (not just accuracy!)

- Multiplicative effect of improvements!
  - Great hardware advances…
  - Better compilers
  - Optimize the core
  - Provide/optimize useful functionality
  - Take full advantage of future improvements at any level



Implementation of ACML

- 32- and 64-bit versions
- PGI and GNU compiler versions
- Linux and Windows (Windows 64-bit to come)
- Hand-coded assembly language kernels
  - Assembly for many BLAS and FFT routines (LAPACK benefits from efficient BLAS kernels)
  - Use Streaming SIMD Extension (SSE vectorized instructions)
- All routines have test programs

ACML 1.0 contents

- BLAS – Level 1, 2 and 3
- LAPACK – all routines and drivers from LAPACK 3
- FFTs – a selection of routines
- All routines with FORTRAN interface

In all, ACML 1.0 contains approximately 900 user-callable routines.

ACML 1.5 highlights

- Key LAPACK routines upgraded to scale for multi-processor machines (PGI OpenMP compiler)
- All routines now have C as well as FORTRAN interfaces
- 32-bit BLAS assembly kernels upgraded
- New 32-bit ACML versions will run on older processors lacking complete set of SSE instructions
- Added 24 Level 1 sparse BLAS routines

ACML C interfaces

- ACML 1.5 contains FORTRAN and C interfaces to all routines
- C interfaces have different argument types and no need for workspace arguments
  - scalar input arguments passed by value, not reference
  - functions allocate their own memory

e.g. FORTRAN:

```
SUBROUTINE DGEQRF( M, N, A, LDA, TAU,
                                 WORK, LWORK, INFO )
```

and C:

```
void dgeqrf (m, n, a, lda, tau, info);
```

ACML Tuning

- Use vectorized SSE instructions

  e.g. mulps %xmm1, %xmm2

  xmm1 and xmm2 are 128-bit registers, each holding four 32-bit floating-point numbers. The mulps instruction multiplies them elementwise:

xmm1	a1	a2	a3	a4
xmm2	b1	b2	b3	b4
xmm1*xmm2	a1*b1	a2*b2	a3*b3	a4*b4

- Arrange data so it can be operated on as above – cleanup loops are likely to be required

ACML Tuning (continued)

- Use prefetching where appropriate
  - Ensure that data is loaded into cache memory before it is used
  - For performance, arrange algorithm so that everything in cache is used – not just part of it
  - AMD64 chips have hardware prefetching, which can be hurt by explicit prefetching – need to take care
- Unroll loops where possible
- Arrange data loads and stores to minimize processor stalling
- Experiment to choose optimal data storage

ACML Tuning (continued)

- Algorithms used by LAPACK routines completely redesigned to take maximum advantage of multi-processor SMP systems
  - New blocking algorithms designed from scratch
  - Easily outperforms standard (netlib) LAPACK
  - No compromise of performance on single processor
- Automatic procedure to choose best block sizes
- AMD64 Software Optimization Guide available:

  http://www.amd.com/us-en/Processors/TechnicalResources/

Making best use of ACML

- Search for opportunities to use ACML routines in your code
  - e.g. many packages use ad-hoc linear equation solvers or singular value decomposition
- If possible, use optimal problem dimensions
  - e.g. FFTs work best on sequences of length $n = 2^k$
  - Blocked LAPACK algorithms work best when problem size is exact multiple of block size – no cleanup
- Use OMP_NUM_THREADS to control number of processors on SMP machines

ACML Library Testing

- During library implementation, each routine must be verified to work correctly. We use stringent test programs.
- Correctness of all user-callable routines must be verified by a stringent test. Ideally, all possible paths through code should be tested.

Anatomy of a stringent test

Example: *DPOTRF* – Cholesky factorization

Given a square, symmetric positive-definite matrix *A*, find lower triangular matrix *L* or upper triangular matrix *U* such that

$$A = LL^T \qquad \text{or} \qquad A = U^T U$$

e.g.

$$A = \begin{bmatrix} 25 & 15 \\ 15 & 18 \end{bmatrix} \quad L = \begin{bmatrix} 5 & 0 \\ 3 & 3 \end{bmatrix} \quad L^T = \begin{bmatrix} 5 & 3 \\ 0 & 3 \end{bmatrix}$$

DPOTRF Stringent Test

SUBROUTINE DPOTRF(UPLO, N, A, LDA, INFO)

- *UPLO* specifies whether *U* or *L* matrix required
- *N* is the order of matrix *A*
- *LDA* is the leading dimension of the array holding *A*
- *INFO* is the error indicator

DPOTRF Stringent Test (cont.)

(1) Tests of illegal arguments:

- *UPLO* not equal to *'U'* or *'L'*
- *N < 0*
- *LDA < N*

(2) Tests of trivial matrices:

- *N = 0* (check no error exit occurred)
- *N = 1* (check that *L* or *U* is equal to sqrt(*A*))
- *N = 1* with negative element in *A* (check that *INFO* is correctly set)

DPOTRF Stringent Test (cont.)

(3) Tests of small matrices with integer elements; essentially exact *L* or *U* expected.

(4) Test of a non-positive-definite matrix *A*.

(5) Test of matrices with increasingly high condition number.

- In all cases, check that:
  - *INFO* returned as expected
  - No input-only arguments were unduly modified
  - Returned matrix *L* or *U* satisfies equation to required tolerance

Test Matrix Generation (1)

For “exact” problems:

- Generate a lower triangular matrix *L* with small integer elements
- Multiply *L* by L^T to create the test matrix *A.*
- Test that the matrix *L* returned by DPOTRF satisfies the condition

where n is the problem size and ε is the arithmetic precision. In addition, check that the individual elements of A are close to the exact integers.

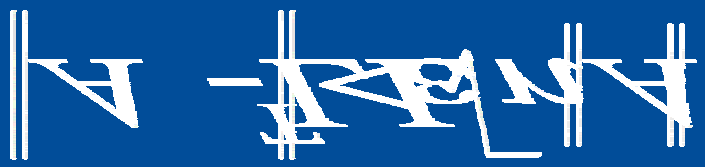
Test Matrix Generation (2)

For problems of predetermined condition number:

- Generate a diagonal matrix of required condition number κ:

$$D = \begin{bmatrix} 0 & \cdots & 0 & \kappa \\ \vdots & & \iddots & 0 \\ 0 & \iddots & & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}$$

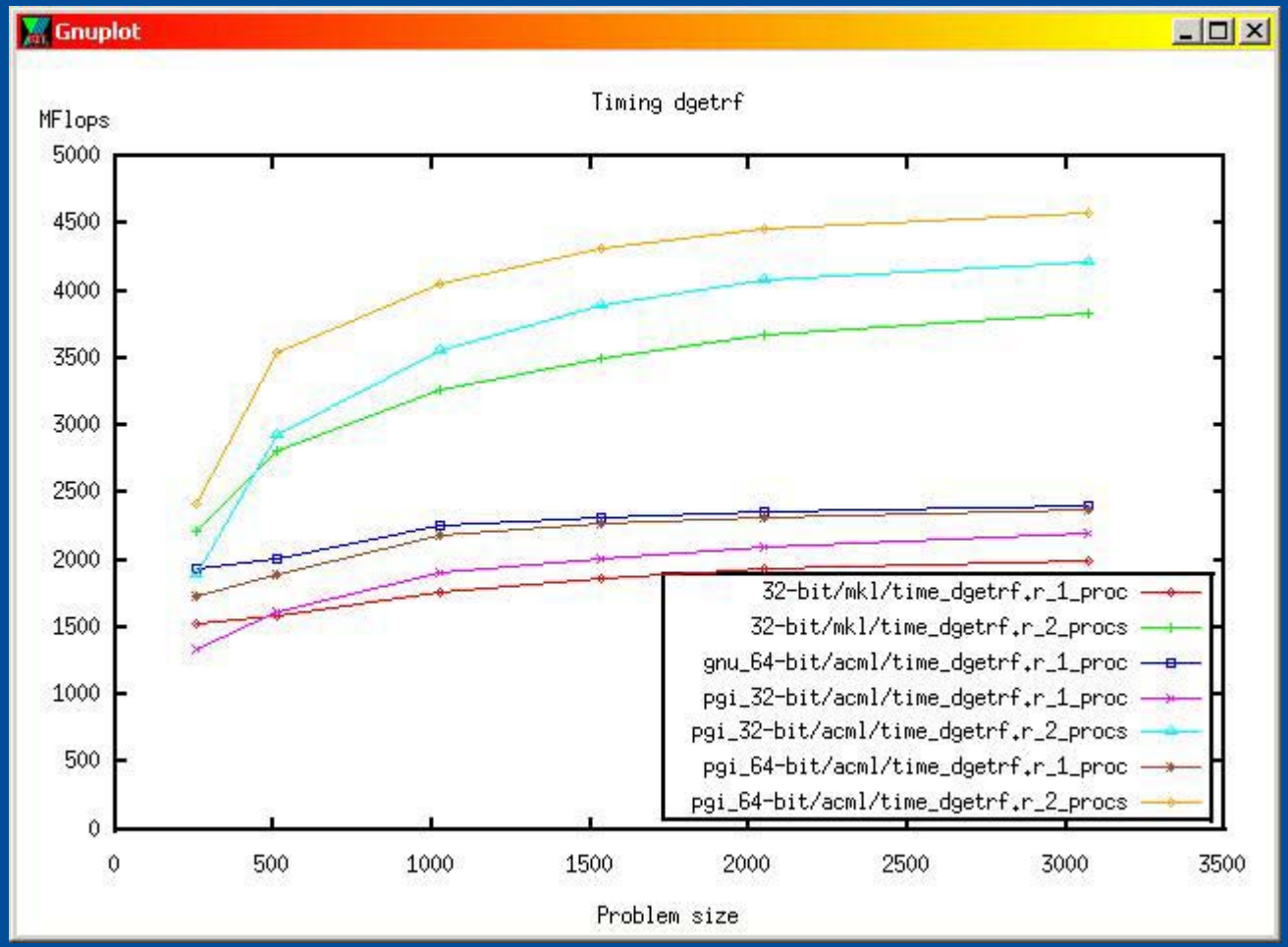
(The elements of *D* are its *singular values* and all are positive)

- Generate a random orthogonal matrix *U,* by means of Householder transformations.
- Create $A = U\,D\,U^T$ - note that A is symmetric and positive-definite.
- Again, check 

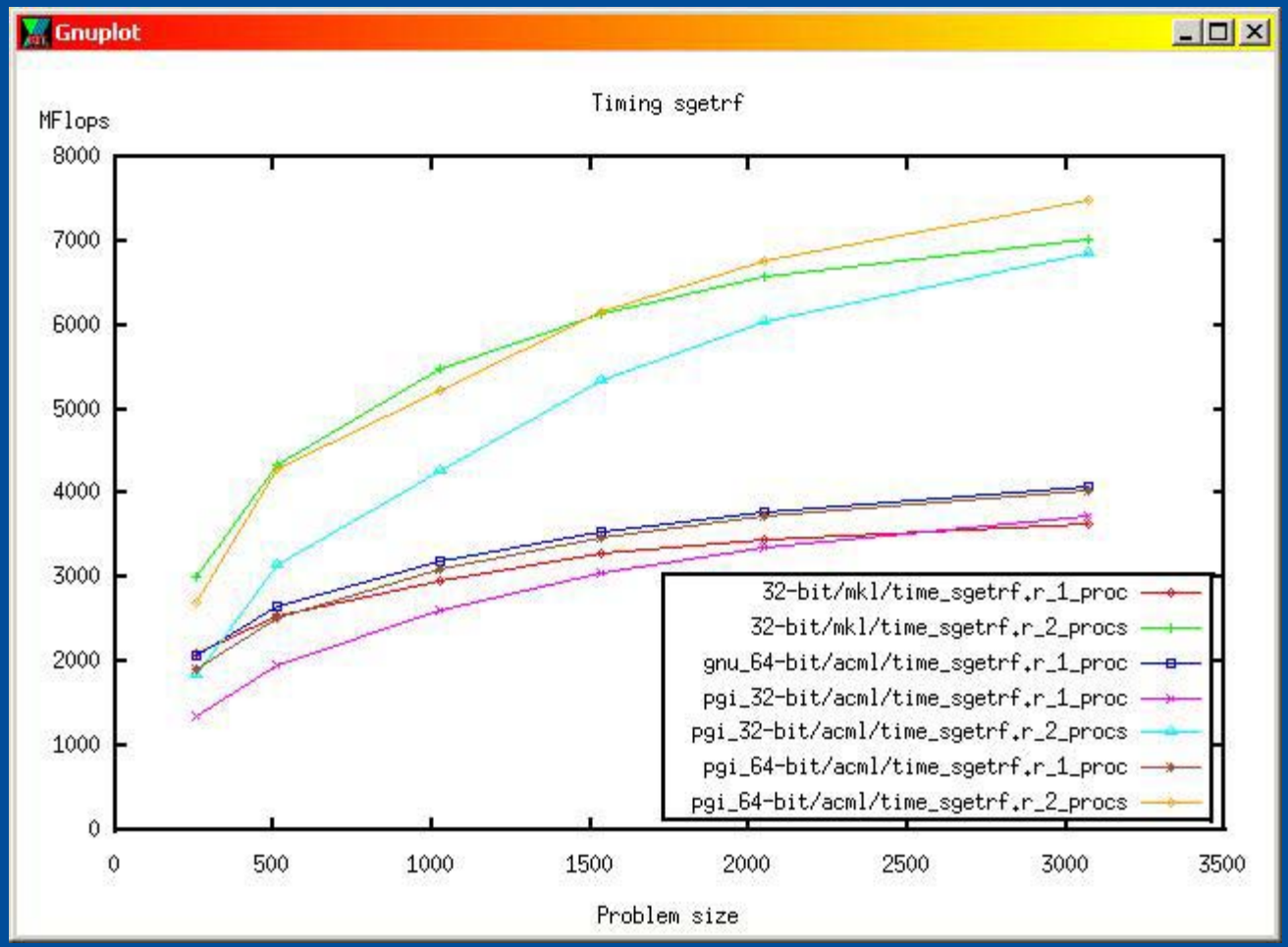
Some Timing Results

- Comparing ACML 1.5 with Intel® MKL 5.2 (32-bit)
- Results generated on dual processor 1600 MHz pre-release AMD Opteron™ system, 1 MB cache size

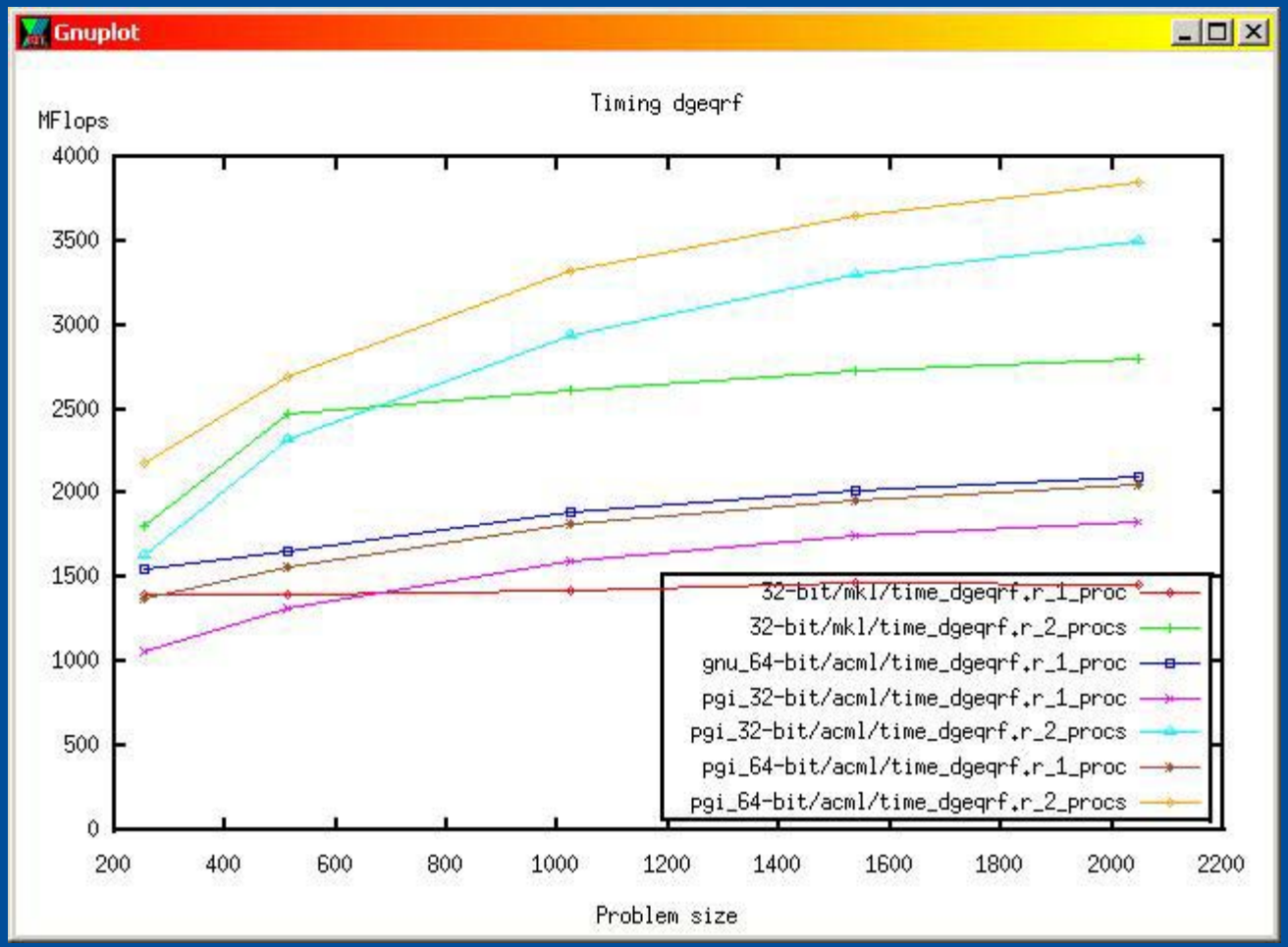
DGETRF – LU factorization



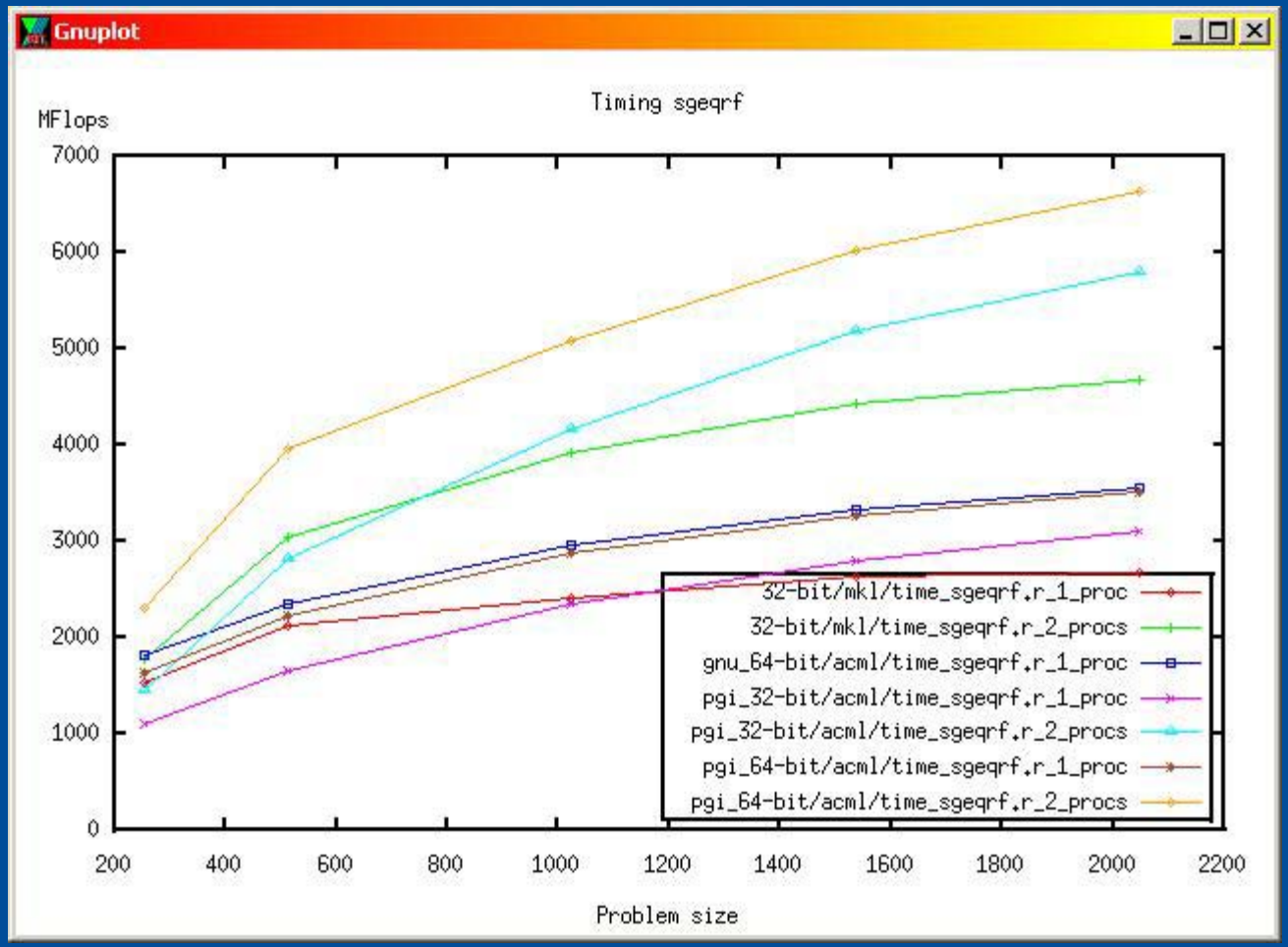
SGETRF – LU factorization



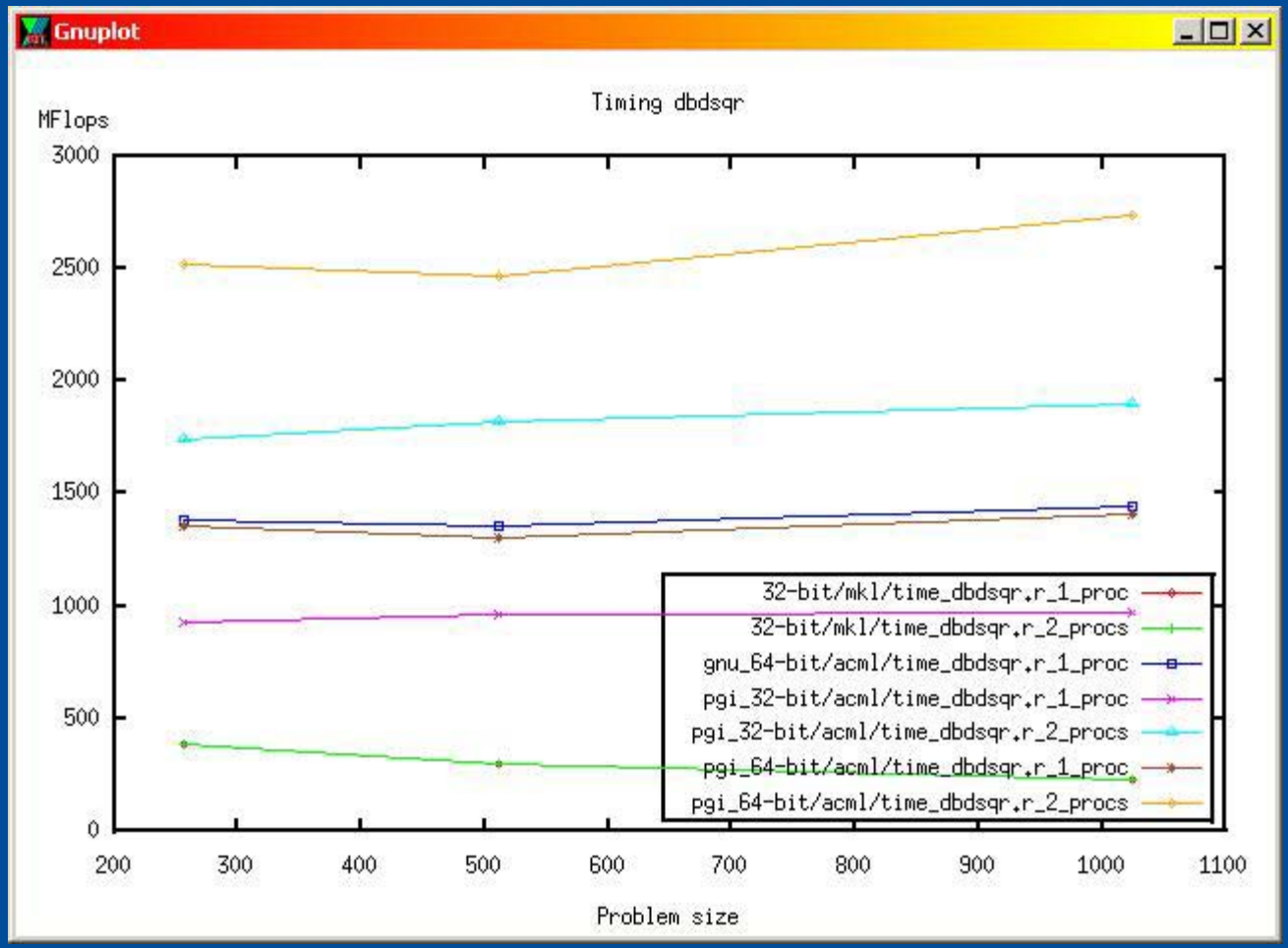
DGEQRF – QR factorization



SGEQRF – QR factorization



DBDSQR – SVD of a bidiagonal matrix



Other AMD/NAG collaboration - standard math library (libm)

- Contains low-level mathematical functions, e.g.:
  - Trigonometry: sin(x), cos(x), tan(x), asin(x), etc.
  - Hyperbolic: sinh(x), cosh(x), tanh(x)
  - Exponential: exp(x), exp10(x), log(x), log10(x) etc.
  - Truncation: ceil(x), floor(x)
  - Power functions: pow(x), pow10(x)
- Linked by C, Fortran programs
- Clearly, libm must be efficient and accurate

Requirements for libm Project

- Linux (for glibc)
- Windows
- Test harness
- Timing harness
- Fast (beat existing 32-bit code)
- Accurate (target error < 1 ulp)

What's an ULP?

- ULP = Unit in the Last Place - a relative measure
- The value of an ULP depends on the size of the number. In IEEE double precision:
  - An ULP of the number 1.0 is worth 2^{-52} (i.e. ~2.22e-16)
  - An ULP of 2.0 is worth double that, and so on
- Target accuracy < 1 ULP means all bits of result must be accurate except perhaps the last one

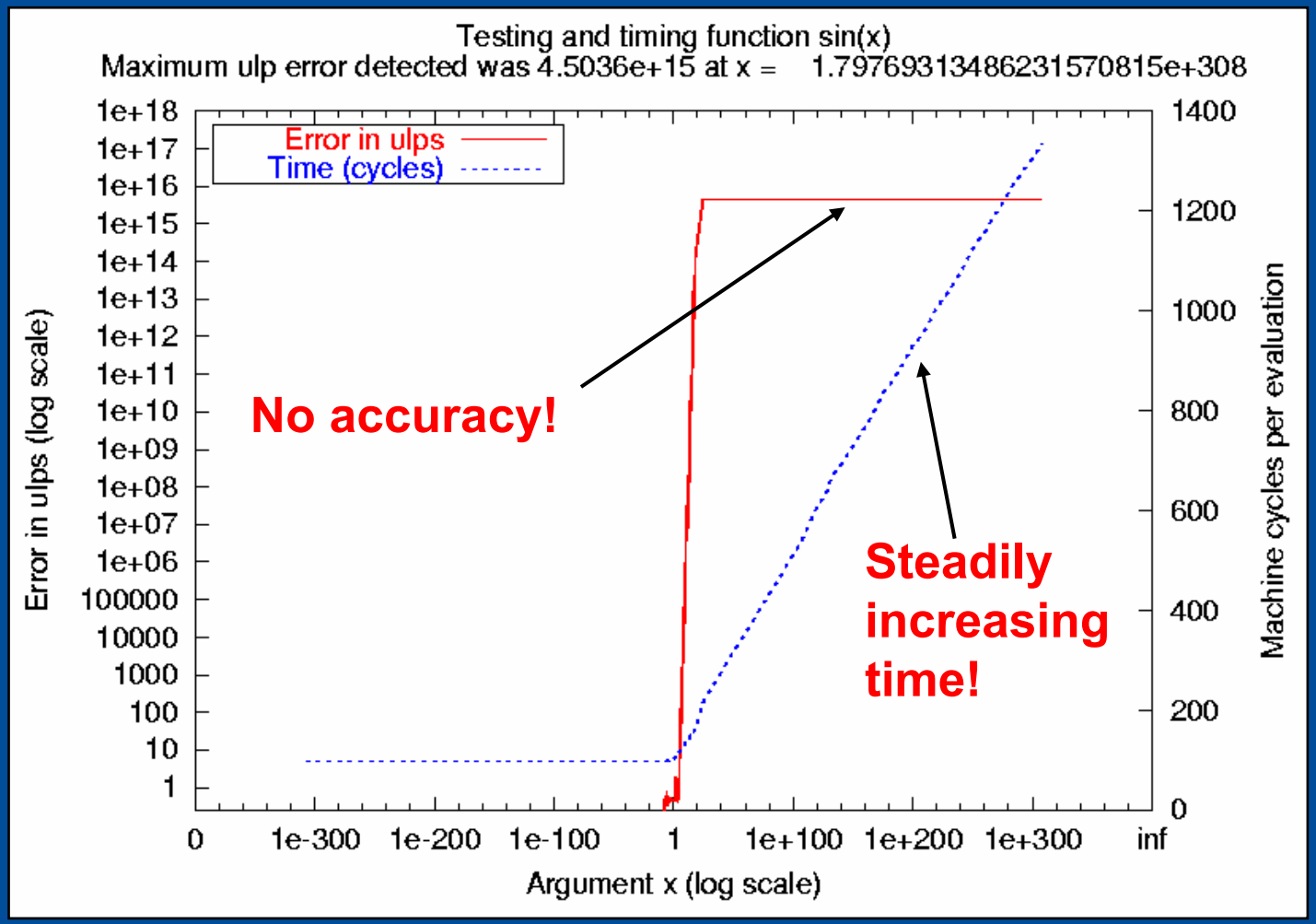
libm Test Harness Reference Implementation

- FPV - Floating-Point Validation package for basic operations + - * / √ (NAG/NPL, 1985)
- Strategy – modify FPV:
  - Basic operations simulated using integer arithmetic to any precision
  - Higher level functions sin(x) etc. built on top of basic operations
  - Simple, slow but accurate algorithms, e.g.Taylor series

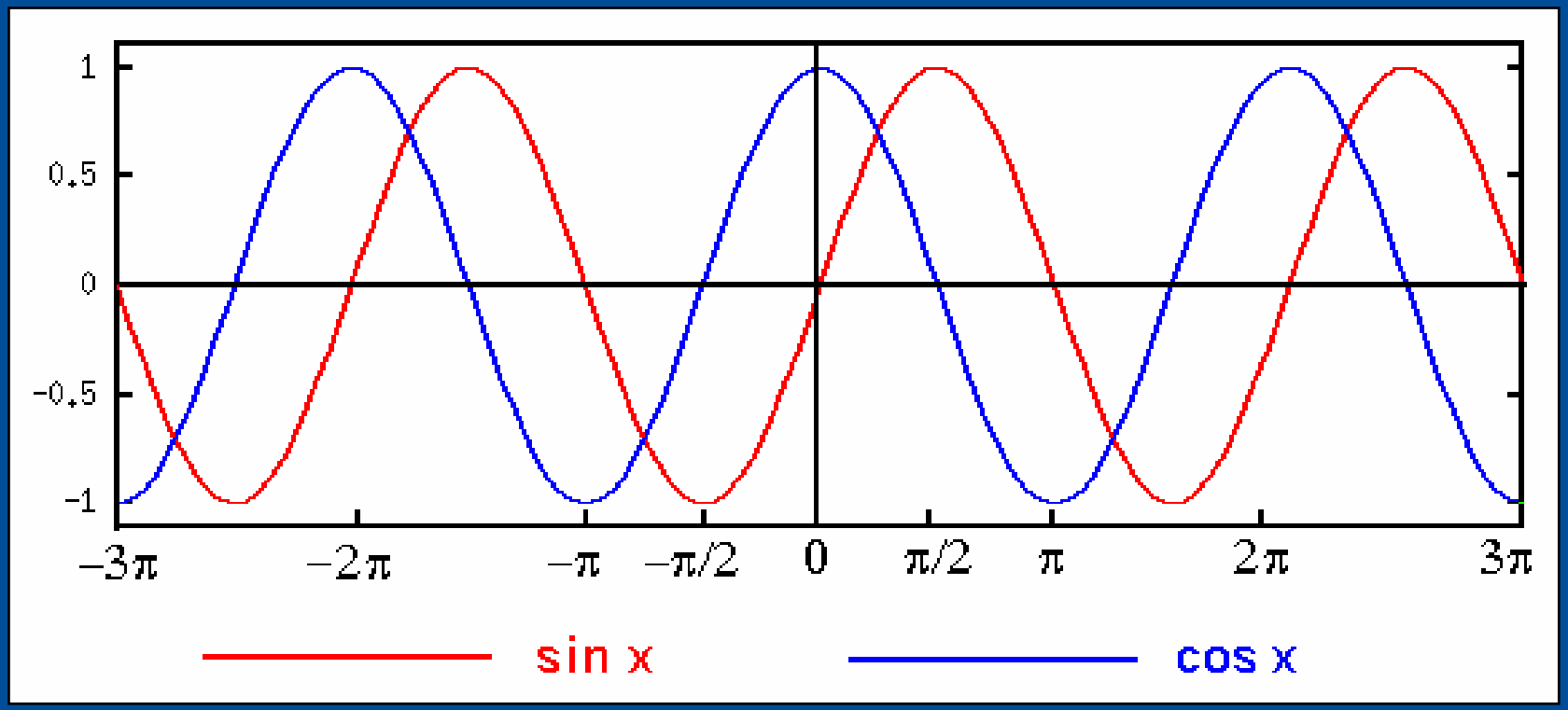
libm Test Harness (cont.)

- Once multi-precision routines are written:
  - Choose intervals of interest, over entire range
  - For a set of numbers in each interval, compare against simulated multi-precision (reference) result
  - Save maximum ULP error
  - Use Maple http://www.maplesoft.com/ as independent verification of max error (test the test!)
  - Also keep track of time in machine cycles

32-bit sin(x) on AMD Opteron™ (old glibc)



Periodicity of sin(x) and cos(x)



$\sin(x) = \sin(x + 2\pi) = \sin(x + 4\pi) = \ldots$

$\sin(x) = \cos(x - \pi/2)$

Argument Reduction

- Primary range is [-π/4, π/4]. If x does not lie in that range:
  - Reduce argument x into r in primary range by subtracting correct multiple of π/2
  - Depending on the multiple, evaluate sin(x) as sin(r) or cos(r) with sign adjusted
  - For r in primary range, evaluate using a fast polynomial approximation

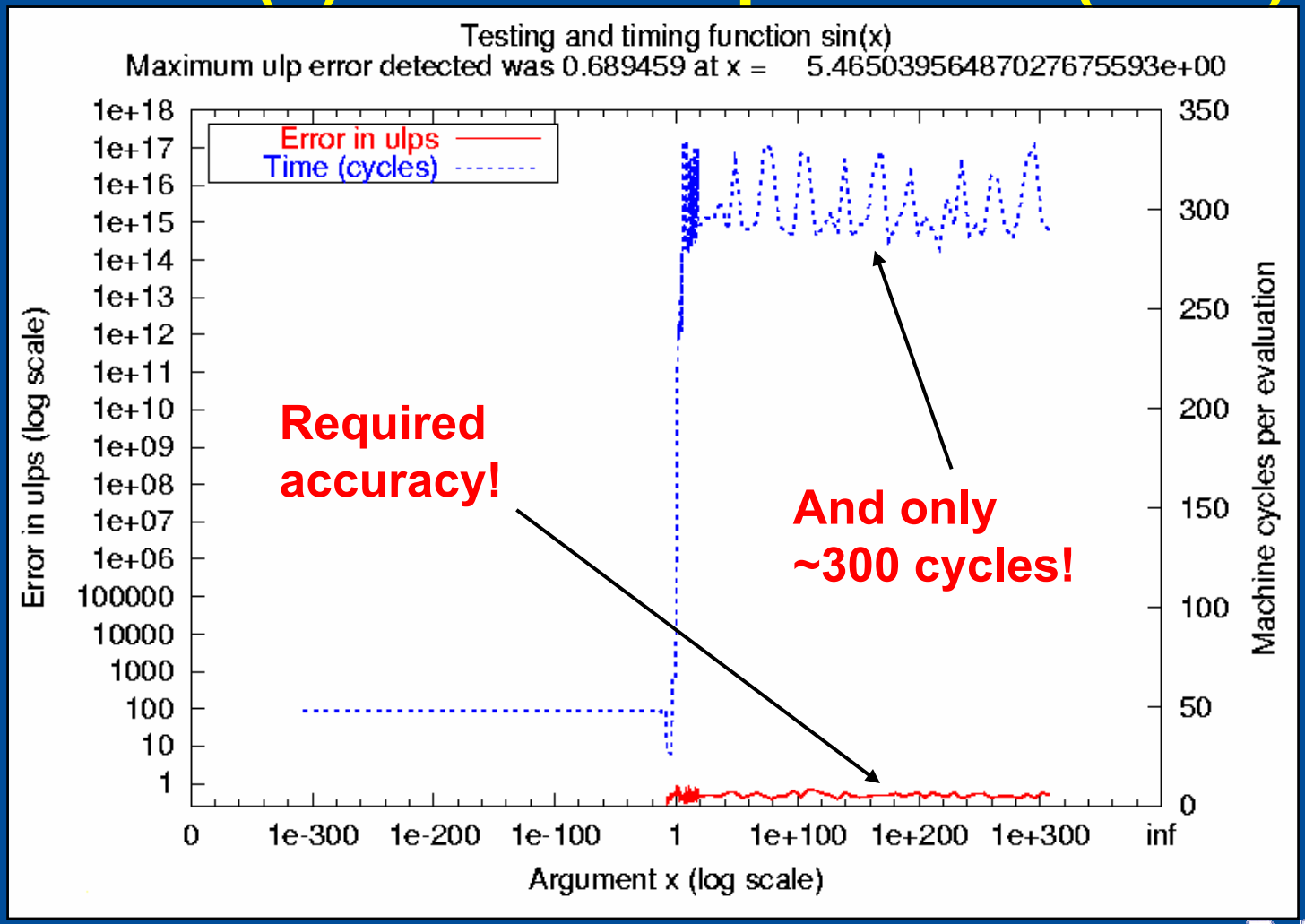
Argument Reduction

- Problem: π is not a representable number
  - finding the right multiple is difficult, especially when x is large – be careful of catastrophic cancellation
- Solution is to store $2/\pi$ in extra precision
  - For IEEE double precision arithmetic, about 500 bits are adequate
  - Payne and Hanek, Radian reduction for trigonometric functions, SIGNUM Newsletter 18:19-24, 1983
- We took advantage of AMD Opteron™ 64-bit integer arithmetic for efficiency in the extended precision multiplication

Implementation

- In general, we use a mixture of:
  - Table-driven algorithms
  - Argument reduction
  - Chebyshev approximations for kernels
  - Occasional assembly language if necessary
- No x87 instructions
  - Tough – especially Remainder!
- No 80-bit registers
- The test harness keeps us honest!

64-bit sin(x) on AMD Opteron™ (NAG)



Lessons Learned

- New architectures bring new opportunities … and new challenges
  - No 80-bit extended precision registers
  - 64-bit integer arithmetic
  - Compilers don't always handle vectorized SSE instructions well – hence hand-coded assembly essential for performance

Summary

- Math libraries are important for ongoing performance improvements – and need to be accurate!
  - How fast do you want the wrong answer?
  - High performance and accuracy are not always mutually exclusive!
- A good Test Harness and related tools are essential
- Use of highly tuned assembly is sometimes critical:
  - For single precision code, Fortran compilers cannot compete with SSE2 (streaming SIMD) vectorized instructions.
  - But time consuming and difficult to maintain
- AMD innovation and AMD / NAG collaboration has helped create a superior platform